**CS422/522 Lab 2: Container & Virtual Memory Management**
due 2015-10-06



# CS422/522 Lab 2: Container & Virtual Memory Management

due 2015-10-06

---

# Introduction

This lab is split into two parts. The first small part continues the work on the physical memory management in the last assignment by implementing a container management layer on top. The second part, which is the main part of this assignment, focuses on the implementation of a page-based virtual memory management of mCertiKOS. You will modify mCertiKOS to set up the page structures for the memory management unit (MMU) of CPU according to the specifications we provide.

## Getting started

In this and future labs you will progressively build up your kernel. We will also provide you with some additional source. To fetch that source, use Git to commit changes you've made since handing in lab 1 (if any), fetch the latest version of the course repository. Then create a local branch called `lab2` based on our lab2 branch, `origin/lab2`:

```
$ cd ~/cpsc422/mcertikos
$ /c/cs422/apps/syncrepo.sh
remote: Counting objects: 130, done.
remote: Compressing objects: 100% (70/70), done.
remote: Total 110 (delta 39), reused 110 (delta 39)
Receiving objects: 100% (110/110), 37.54 KiB | 0 bytes/s, done.
Resolving deltas: 100% (39/39), completed with 17 local objects.
From /c/cs422/repo/mcertikos
 * [new branch]      lab2       -> lab2
****
Remote repository synchronized successfully.
****
$ git pull
remote: Counting objects: 130, done.
remote: Compressing objects: 100% (70/70), done.
```

```
remote: Total 110 (delta 39), reused 110 (delta 39)
Receiving objects: 100% (110/110), 37.54 KiB | 0 bytes/s, done.
Resolving deltas: 100% (39/39), completed with 17 local objects.
From /c/cs422/SUBMIT/lab/xw247
 * [new branch]      lab2       -> origin/lab2
Already up-to-date.
$ git checkout -b lab2 origin/lab2
Branch lab2 set up to track remote branch refs/remotes/origin/lab2.
Switched to a new branch "lab2"
$
```

You will now need to merge the changes you made in your `lab1` branch into the `lab2` branch, as follows:

```
$ git merge lab1
Merge made by recursive.
 kern/pmm/MATIntro/MATIntro.c  |   11 +++++++++--
 kern/pmm/MATInit/MATInit.c    |   19 ++++++++++++++++++
 kern/pmm/MATOp/MATOp.c        |    7 +++----
 3 files changed, 31 insertions(+), 6 deletions(-)
$
```

In some cases, Git may not be able to figure out how to merge your changes with the new lab assignment (e.g. if you modified some of the code that is changed in the second lab assignment). In that case, the git merge command will tell you which files are *conflicted*, and you should first resolve the conflict (by editing the relevant files) and then commit the resulting files with `git commit -a`.

If you have trouble merging the branches, another simple solution is to copy the files you have changed in the assignment 1 into appropriate directories of assignment 2, overwriting the existing files. You should copy at least the following files:

- `kern/pmm/MATIntro/MATIntro.c`

- `kern/pmm/MATInit/MATInit.c`
- `kern/pmm/MATOp/MATOp.c`

It is possible that you may fail the tests for this assignment because of a bug existed in your code for assignment 1. We will release sample solution code for assignment 1 by the end of this week (three days after its official due date). The code will be provided in a directory called `samples` under the root project directory. Please feel free to use the sample code instead of your own code. When the sample code is released, commit your local changes, and then retrive the sample code by:

```
1.  $ cd ~/cpsc422/mcertikos
2.  $ /c/cs422/apps/syncrepo.sh
3.  ****
4.  Remote repository synchronized successfully.
5.  ****
6.  $ git pull
7.  $ git push
```

* Once you've synced your remote repository, you have to run `git push` again to push your local changes back to the remote repository, since the remote one is forced to be synchronized with the updated version.

# Hand-In Procedure

Include the following information in the file called `README` in the `mcertikos` directory: who you have worked with; whether you coded this assignment together, and if not, who worked on which part; and a brief description of what you have implemented; and anything else you would like us to know. When you are ready to hand in your lab code, commit your changes, and then run `git push` in the `mcertikos`.

```
1.  $ git commit -am "finished lab2"
2.  [lab2 a823de9] finished lab2
3.   4 files changed, 87 insertions(+), 10 deletions(-)
```

```
4.  $ git push
```

# Part 1: Physical Memory Management, Continued

## Introducing the Container

In mCertiKOS, a container is an object used to keep track of the resource usage of each process, as well as the parent/child relationships between processes. It is important for the kernel to track resource usage so that malicious processes can be prevented from using up all the available resources, resulting in a denial-of-service attack. In mCertiKOS, if a user process attempts to allocate all available memory (e.g., by calling malloc in an infinite loop), the kernel will deny all allocation requests once the process has allocated its maximum allowed quota.

Note that the only resource we currently track is the number of pages allocated by each process. However, we designed the container mechanism in a general way so that we can easily extend it to track other types of resources. One interesting example (and potential research project) would be extending containers to track CPU time as a resource.

To describe containers in more detail, we first need to define a way to distinguish a particular process. We do this via unique IDs. Whenever a process is spawned, it is assigned an unused ID in some range [0, NUM_IDS). Every ID has an associated container. ID 0 is reserved for the kernel itself.

When a new ID is created, how do we decide on the maximum quota passed to that ID? One possible solution is to fix some specific max quota for all IDs. This is quite restrictive, however, since some programs may require vastly different resource usage than others. In mCertiKOS, we define a parent/child relationship between IDs, and we require that parents choose resources to pass on to their children. Each ID has a single parent, and potentially multiple children. ID 0 is called the "root", as it is the root of the parent/child tree and thus the only container without a parent.

Consider any ID i. The fields of i's container are as follows:

- `quota` - the maximum number of pages that ID i is allowed to use
- `usage` - the number of pages that ID i has currently allocated for itself or distributed to children
- `parent` - the ID of the parent of i (or 0 if i = 0)
- `nchildren` - the number of children of i
- `used` - a boolean saying whether or not ID i is in use (if this boolean is false, then ID i is not in use and the values of the other fields of container i should be ignored)

During the execution of mCertiKOS, there are two situations where containers will be used:

1. whenever a page allocation request is made (e.g., handling a page fault or handling a malloc system call request); and
2. whenever a new ID is spawned (the parent ID must distribute some of its quota to the newly-spawned child).

To reason about the relationships between the container objects and the actual available resources, the mCertiKOS kernel must maintain the following invariant throughout execution.

**Soundness**: The sum of the available quotas (i.e., quota minus usage) of all used IDs is at most the number of pages available for allocation.

When writing code to implement containers, be sure that the initialization primitive establishes this invariant, and each other primitive maintains it.

---

# The MContainer Layer

In this layer, you are going to implement various functions to maintain the containers used in mCertiKOS. Please make sure you read all the comments carefully.

> ## Exercise 1

In the file `kern/pmm/MContainer/MContainer.c`, you must implement all the functions listed below:

- `container_init`
- `container_get_parent`
- `container_get_nchildren`
- `container_get_quota`
- `container_get_usage`
- `container_can_consume`
- `container_split`
- `container_alloc`
- `container_free`

---

## Testing The Kernel

We will be grading your code with a set of test cases, part of which are given in `test.c` in each layer sub directory. You can run `make TEST=1` to test your solutions. You can use `Ctrl-a x` to exit from the qemu.

* If you have already run `make` before, you have to first run `make clean` before you run `make TEST=1`.

```
1.  Testing the MContainer layer...
2.  test 1 passed.
3.  test 2 passed.
4.  All tests passed.
5.
6.  Testing the MPTIntro layer...
7.  test 1 passed.
8.  test 2 passed.
9.  All tests passed.
10.
```

```
Testing the MPTOp layer...
test 1 passed.
All tests passed.

Testing the MPTComm layer...
test 1 passed.
test 2 passed.
All tests passed.

Testing the MPTKern layer...
test 1 passed.
test 2 passed.
All tests passed.

Testing the MPTNew layer...
test 1 passed.
All tests passed.

Test complete. Please Use Ctrl-a x to exit qemu.
```

Make sure your code passes all the tests for the `MContainer` layer.

## Write Your Own Test Cases! (optional)

Come up with your own interesting test cases to seriously challenge your classmates! In addition to the provided simple tests, selected (correct, fully documented, and interesting) test cases will be used in the actual grading of the lab assignment!

In `test.c` in each layer directory, you will find a function defined with the name `LayerName_test_own`. Fill the function body with all of your nice test cases combined. The test function should return 0 for passing the test and a non-zero code for failing the test. Be extra careful to make sure that if you overwrite some of the kernel data, they will be set back to the original value. Otherwise, it may make the future test scripts to fail even if you

implement all the functions correctly.

* Your test function itself will not be graded. So don't be afraid of submitting a wrong script.

---

# Part 2: Virtual Memory Management

Before doing anything else, familiarize yourself with the x86's protected-mode memory management architecture: namely *segmentation* and *page translation*.

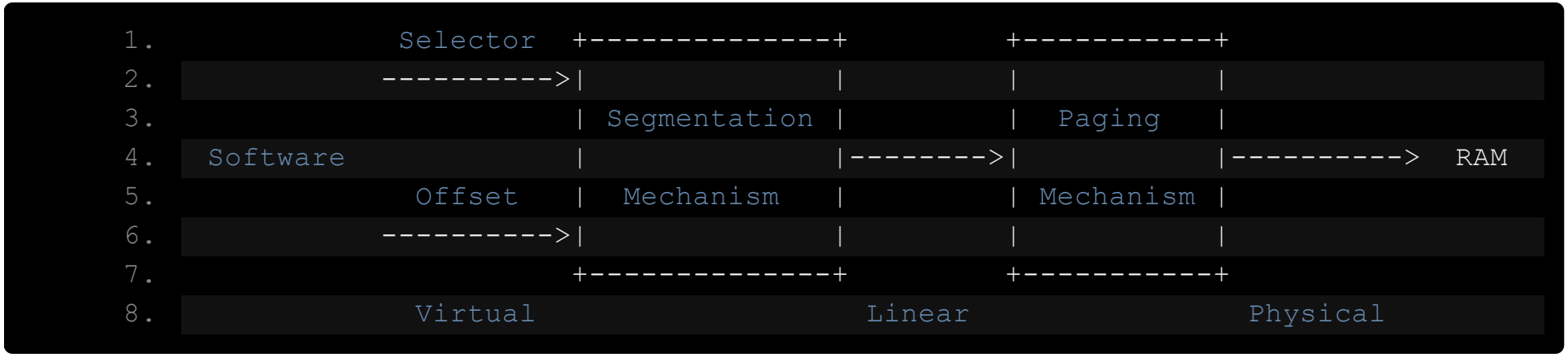
> **Exercise 2**
>
> Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while mCertiKOS uses paging for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.
>
> Please read the above document carefully and make sure that you understand how the page map is structured and how virtual memory works on the Intel x86 platforms. As a typical advanced computer science class, we will no longer provide very detailed step by step guide on how to implement each function in each layer. You are expected to carefully review related documents and walk through different parts of the kernel code to figure out some details by yourself (to have some real operating system hacking experience). The functions are well documented. If any of them does not make sense to you, feel free to post a question on piazza.

* This part requires significantly more amount of effort comparing to the previous assignment. Please start early!

segment translation
-> linear address
|
page translation
-> physical address

# Virtual, Linear, and Physical Addresses

In x86 terminology, a *virtual address* consists of a segment selector and an offset within the segment. A *linear address* is what you get after segment translation but before page translation. A *physical address* is what you finally get after both segment and page translation and what ultimately goes out on the hardware bus to your RAM.




A C pointer is the "offset" component of the virtual address. In mCertiKOS, we installed a Global Descriptor Table (GDT) that effectively disabled segment translation by setting all segment base addresses to 0 and limits to `0xffffffff`. Hence the "selector" has no effect and the linear address always equals the offset of the virtual address. In lab 3, we'll have to interact a little more with segmentation to set up privilege levels, but as for memory translation, we can ignore segmentation throughout the mCertiKOS labs and focus solely on page translation.

Before lab 3

### Exercise 3

While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU monitor commands from the lab tools guide, especially the `xp` command, which lets you inspect physical memory. To access the QEMU monitor, press Ctrl-a

> c in the terminal (the same binding returns to the serial console).
>
> Use the xp command in the QEMU monitor and the x command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data. You do not need to submit anything for this exercise.

From code executing on the CPU, once we're in protected mode and the paging is turned on, there's no way to directly use a linear or physical address. *All* memory references are interpreted as virtual addresses and translated by the MMU, which means all pointers in C are virtual addresses.

The mCertiKOS kernel often needs to manipulate addresses as opaque values or as integers, without dereferencing them, for example in the physical memory allocator. The kernel also often needs to treat an integer as an address or a pointer. If you do not understand the C pointer very well, please spend some time to study it carefully before you get started on this lab.

The kernel sometimes also needs to read or modify memory for which it knows only the physical address. For example, adding a mapping to a page structure may require allocating physical memory to store a page directory and then initializing that memory. However, the kernel, like any other software, cannot bypass virtual memory translation and thus cannot directly load and store to physical addresses. In mCertiKOS, we use separate page structure for each process, and we switch page structures when we switch among different processes. To solve the issue above, we reserve the entire page structure with index `0` for the kernel, i.e., the process 0 is always kernel process. Then we configure the entire page structure 0 as the identity map. This way, whenever the kernel needs to access a physical address, we can switch to the page structure 0, and then access whatever physical address (which is the same as the virtual address) we want. In mCertiKOS, we need to initialize many page table mappings as the identity map, i.e., the entire page structure 0, and the kernel portion of the memory for the rest of page structures. Instead of repeatedly allocating the same identity second level page tables, we staticly allocate one and point every appropriate page directory index to the same page table entry. (see `IDPTbl` in the `MPTIntro` layer).

You will implement various parts of the virtual memory management module strictly following the abstraction layers

that we have built for you. Inside the `kern` directory, you will see a sub directory called `vmm`. This is where all the code related to virtual memory management reside. The virtual memory management is devided into six abstraction layers, which corresponds to the further sub directories you see under `vmm`. You will need to implement layer by layer (except MPTInit, which is already fully implemented), from bottom up, following the instructions. Each layer directory contains the following three files:

- `import.h:` The list of functions that are exposed to the current layer are declared and documented here. You are supposed to implement the layer functions using only the functions declared in import.h. This way, you do not have to look at the lower layers to figure out all the details.
- `LayerName.c` The list of functions in the current layer are implemented here. You are supposed to fill in the part marked as `TODO`.
- `export.h` The declarations of the functions of the current layer that are exposed to the upper layers.

---

# Running a Process with Virtual Memory

To better illustrate the process of virtual memory and address translation, we have created an extra command in our kernel monitor called `runproc`. Once run, that command will start a user process defined in `user/proc/dummy/dummy.c`. The process implements a program that allows you to input an arbitrary virtual address to read from or write to. The program is rather limited, and only allows you to enter the address in decimal numbers. Feel free to replace it with whatever fancy programs that you can come up with to test the virtual memory. If you read the current code, you may notice that our implemention of `sys_getc` is a little different from the `getchar` in the C standard library. It does not wait when there is no characters pending in the input buffer. Instead, it simply returns 0 (not the character '0'). Thus, you have to implement the waiting logic by yourself. Read the existing code for a reference.

A sample run of the program is pasted below:

```
1.  ****************************************
2.
```

```
Welcome to the mCertiKOS kernel monitor!

****************************************

Type 'help' for a list of commands.
$> runproc
Program 0x0010a004 is loaded.
Welcome to the user process! (Ctrl - Z to exit)

Specify a virtual address to read from or write to.
Enter the address: 3489660928
Address entered: 3489660928
Specify the action: r for read, w for write.
w
Enter the value you want to write to the address.
Value to write (from 0 to 9): 5
Page fault: VA 0xd0000000, errno 0x00000002, page table # 1, EIP 0x40000255.
Successfully wrote the value to the virtual address. You can double check it
with the read command.

Specify a virtual address to read from or write to.
Enter the address: 3489660928
Address entered: 3489660928
Specify the action: r for read, w for write.
r
The value at virtual address 3489660928 is 5.

Specify a virtual address to read from or write to.
Enter the address:
Exiting from the user process.
$>
```

Pay special attention to the line #19 that is in bold italic font. That line is printed from our page fault handler (see `kern/lib/trap.c`). The page fault was triggered because it was trying to access a non-mapped virtual address (address 3489660928, that is 0xd0000000). When a page fault is triggered due to this reason, the page fault handler dynamically allocates a page for the corresponding virtual address and returns back to the instruction that caused the page fault (in this case, it is one of the instructions in the user process).

p> Note that, during the middle of your virtual memory implementation, running the above command could produce many random errors, or the machine may frequently reboot itself. Don't be panic. This is because your virtual memory management layers have not been fully implemented. Once the layers are fully implemented, you should be able to successfully start the user process as shown above.

Since we have not set up the process management code, in this lab, we have to hack the kernel in a way such that it is under an illusion that we have the user process set up and running. However, note that the current "user process" is actually running in ring0 mode. That means you can actually access arbitrary virtual address using the program we provide. So don't be supprised that when you try to write to some address, it crashes the kernel or results in some funny behavior. In the next assignment, we will learn how to build up the process management layers to schedule and run multiple user processes in the ring 3 mode with memory protection.

Ring 0; kernel
Ring 1: device drivers
Ring2: device drivers
ring3: applications

---

# The MPTIntro Layer

In this layer, you are going to implement the getter and setter functions for two data structures used to maintain the processes' page tables. Please make sure you read all the comments carefully.

> ## Exercise 4
>
> In the file `kern/vmm/MPTIntro/MPTIntro.c`, you must implement all the functions listed below:
>
> - `set_pdir_base`
> - `get_pdir_entry`
> - `set_pdir_entry`

- `set_pdir_entry_identity`
- `rmv_pdir_entry`
- `get_ptbl_entry`
- `set_ptbl_entry`
- `set_ptbl_entry_identity`
- `rmv_ptbl_entry`

Make sure your code passes all the tests for the `MPTIntro` layer. And write your own test cases to challenge other students' implementations.

---

# The MPTOp Layer

### Exercise 5

In the file `kern/vmm/MPTOp/MPTOp.c`, you must correctly implement all the functions listed below:

- `get_pdir_entry_by_va`
- `set_pdir_entry_by_va`
- `rmv_pdir_entry_by_va`
- `get_ptbl_entry_by_va`
- `set_ptbl_entry_by_va`
- `rmv_ptbl_entry_by_va`
- `idptbl_init`

Make sure your code passes all the tests for the `MPTOp` layer. And write your own test cases to challenge other students' implementations.

# The MPTComm Layer

> ### Exercise 6
>
> In the file `kern/vmm/MPTComm/MPTComm.c`, you must correctly implement all the functions listed below:
>
> - `pdir_init`
> - `alloc_ptbl`
> - `free_ptbl`

Make sure your code passes all the tests for the `MPTComm` layer. And write your own test cases to challenge other students' implementations.

# The MPTKern Layer

> ### Exercise 7
>
> In the file `kern/vmm/MPTKern/MPTKern.c`, you must correctly implement all the functions listed below:
>
> - `pdir_init_kern`
> - `map_page`
> - `unmap_page`

Make sure your code passes all the tests for the `MPTKern` layer. And write your own test cases to challenge other students' implementations.

# The MPTInit Layer

In this layer, we set the CR3 register to the initial address of the page structure #0, and then turn on the paging. There's no exercise in this layer.

---

# The MPTNew Layer

> ## Exercise 8
>
> In the file `kern/vmm/MPTNew/MPTNew.c`, you must correctly implement all the functions listed below:
>
> - `alloc_page`

Make sure your code passes all the tests for the `MPTNew` layer. And write your own test cases to challenge other students' implementations.

---

**This completes the lab.** Make sure you pass all of the `make TEST=1` tests and don't forget editing your `README` file. Commit your changes and type `git push` in the `mcertikos` directory to submit your code.